Technical Disclosure Commons

Defensive Publications Series

July 03, 2017

LANGUAGE RUNTIME NON-VOLATILE RAM AWARE SWAPPING

Language Runtime Non-Volatile RAM Aware Swapping

Abstract

One possible way to take advantage of NVRAM's lower latency is to use it as swap space for a program. Moving stored data from RAM to NVRAM frees up RAM space that can then be used for storing other program data or instructions. Since data stored in RAM can be accessed faster, moving stored data from RAM to NVRAM increases the overall performance of the program. As an example, a language runtime may take advantage of NVRAM by identifying program memory that is less likely to be used in the near future and storing the identified program memory in NVRAM rather than directly in RAM. This enables the program to consume less RAM, which is per byte more expensive than NVRAM. The advantage of using the language runtime to trigger these types of paging events is that one does not have to wait for the operating system to realize that these memory regions are not being used often.

I. Program Runtime Usage of NVRAM

Non-volatile memory, such as flash memory, retains stored content even when power is turned off. In contrast, volatile memory such as dynamic random access memory (DRAM) and static random access memory (SRAM) loses stored data when power is turned off. Non-volatile random access memory (NVRAM) falls somewhere between flash memory and DRAM. For example, NVRAM has much lower latency than flash memory and it is cheaper than RAM, but it retains data when not powered, much like flash memory.

One possible way to take advantage of NVRAM's lower latency is to use it as swap space for a program. For example, if a program uses large amounts of RAM then some of the program working memory can be stored in NVRAM instead of RAM. Moving stored data from RAM to

NVRAM frees up RAM space that can then be used for storing other program data or instructions. Since data stored in RAM can be accessed faster, moving stored data from RAM to NVRAM increases the overall performance of the program.

A language runtime is a program that manages the execution of a program, sometimes including automatic memory management. For example, a Java runtime manages the execution of a Java program within the Java Virtual Machine. Other programming languages that exploit a runtime program are the languages Go, Dart, C# and Objective C. Figure 1 illustrates a highly simplified diagram of the Java Virtual Machine (JVM). The Java runtime manages the execution of a Java program within the JVM including automatic memory management via a garbage collector. The garbage collector periodically monitors allocation of the program memory, for example, in the heap memory, and deallocates program memory associated with objects that are no longer being referenced by the Java program. The garbage collector looks at the heap memory and identifies objects that are in use and objects that are unused and deletes the unused objects. An object is identified as in use if some portion of the Java program is maintaining a reference to that object. In contrast, an object is identified as unused if no portion of the Java program is maintaining a reference to that object and thus, the memory associated with the unused object can be deallocated.

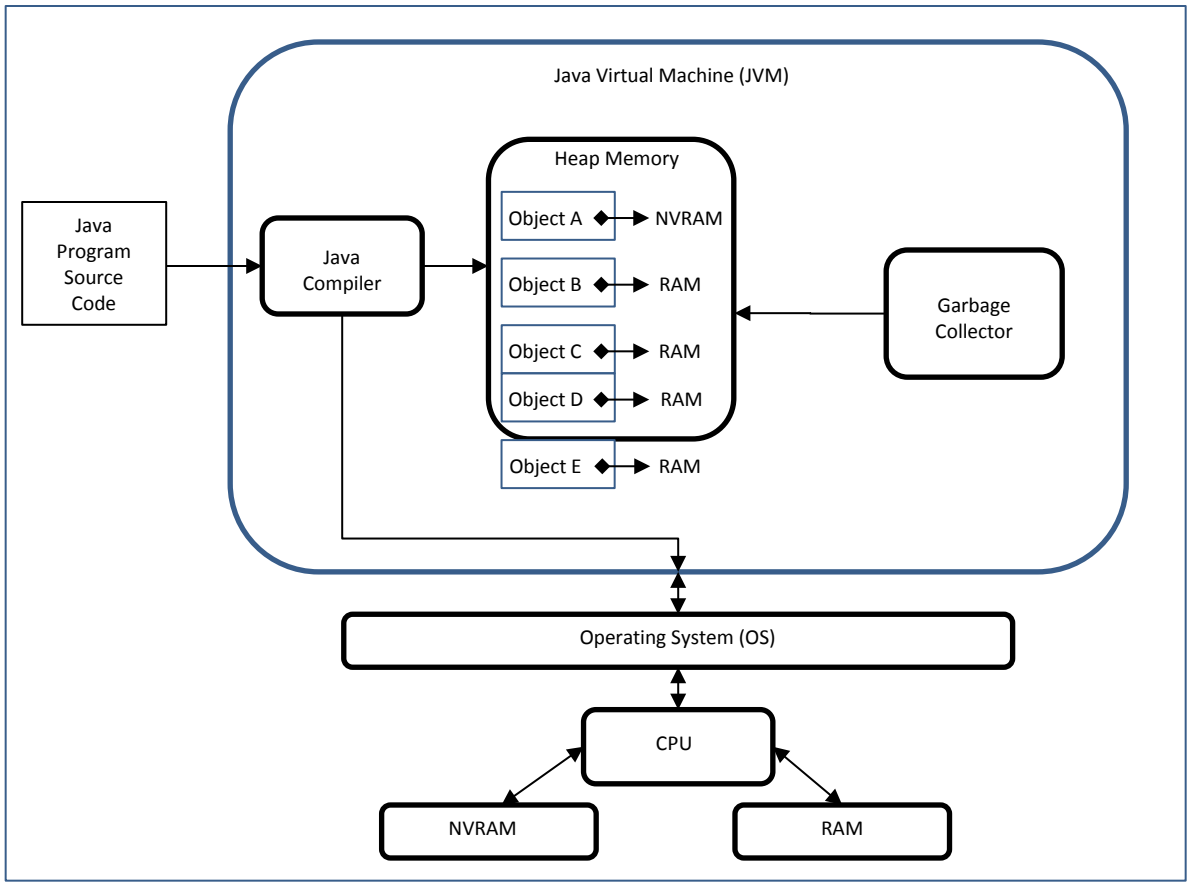

Figure 1: High-level diagram of garbage collection in the JVM.

II. NVRAM Storage Strategies

As an example, a language runtime may take advantage of NVRAM by identifying program memory that is less likely to be used in the near future and storing the identified program memory in NVRAM rather than directly in RAM. Using the language runtime, rather than the operating system, to determine which program memory to store in NVRAM is more efficient because the language runtime, rather than the operating system, has access to more of the information needed to make such a decision. Referring to Figure 1, the garbage collector or other portion of the language runtime can be configured to identify objects in the heap memory that are less likely to be used in the near future and if the identified objects are allocated to be stored in RAM, the objects may be reallocated to NVRAM thereby, moving these objects from

RAM to NVRAM. In Figure 1, Objects B and C may be identified as less likely to be used in the near future by the program and may be reallocated to NVRAM. This enables the program to consume less RAM, which is per byte more expensive than NVRAM. This can be particularly valuable on servers or other computing devices that are simultaneously executing multiple programs. RAM storage requirement savings across multiple applications executing on the computing device can grow to be quite meaningful.

As another example, a garbage collector or automatic memory management program that runs periodically could be configured to identify program memory associated with data structures that will not be needed by the program until the garbage collector's next cycle. The garbage collector could store the identified program memory in NVRAM rather than RAM at the end of the garbage collector's current cycle. This leaves RAM space into which additional program data instructions to be loaded into RAM so that they will be available at lower latency than if they had to be fetched from NVRAM. This increases the overall performance of the program.

As another example, a language runtime could be configured to identify program data that is unlikely to be used in the near future within the program and move the identified data to NVRAM. For example, the Go language runtime could detect memory that is only accessible in Go by using *unsafe* operations (i.e., operations included in the *unsafe* Go package), such as accessing array slices that have been truncated such that they no longer point to the beginning of the underlying array. This memory cannot be freed by the garbage collector but it also cannot be accessed by the language without the use of commands included in the *unsafe* package. This information may be identified as unlikely to be used and can be moved from RAM to NVRAM.

In the case that *unsafe* is used and the information is required, the only cost is a small latency increase associated with retrieving this memory from NVRAM rather than RAM.

As another example, a language runtime may be configured to detect information in memory regions of the program that it knows will not be needed for a long time and move them to NVRAM. For example, in the Go programming language, when a *goroutine* is waiting for a timer to expire, that *goroutine*'s stack or other code that will not be executed until the timer expires can be moved to NVRAM for the duration of the timer. The advantage of using the language runtime to trigger these types of paging events is that one does not have to wait for the operating system to realize that these memory regions are not being used often. Preempting the operating system by leveraging runtime program information can therefore lead to improved performance.